21 | Catalyst逻辑计划：你的SQL语句是怎么被优化的？（上）

2021-04-30 吴磊

Spark性能调优实战 进入课程 >

讲述：吴磊

时长 18:28 大小 16.92M

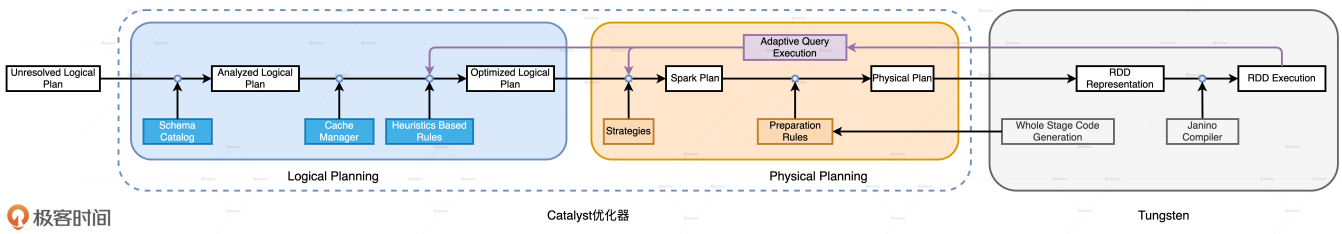
你好，我是吴磊。

上一讲我们说，Spark SQL 已经取代 Spark Core 成为了新一代的内核优化引擎，所有 Spark 子框架都能共享 Spark SQL 带来的性能红利，所以在 Spark 历次发布的新版本中，Spark SQL 占比最大。因此，Spark SQL 的优化过程是我们必须要掌握的。

Spark SQL 端到端的完整优化流程主要包括两个阶段：Catalyst 优化器和 Tungsten。其中，Catalyst 优化器又包含逻辑优化和物理优化两个阶段。为了把开发者的查询优化到极致，整个优化过程的运作机制设计得都很精密，因此我会用三讲的时间带你详细探讨。

下图就是这个过程的完整图示，你可以先通过它对优化流程有一个整体的认知。然后随着我的讲解，逐渐去夯实其中的关键环节、重要步骤和核心知识点，在深入局部优化细节的

同时，把握全局优化流程，做到既见树木、也见森林。



Spark SQL的优化过程

今天这一讲，我们先来说说 Catalyst 优化器逻辑优化阶段的工作原理。

案例：小 Q 变身记

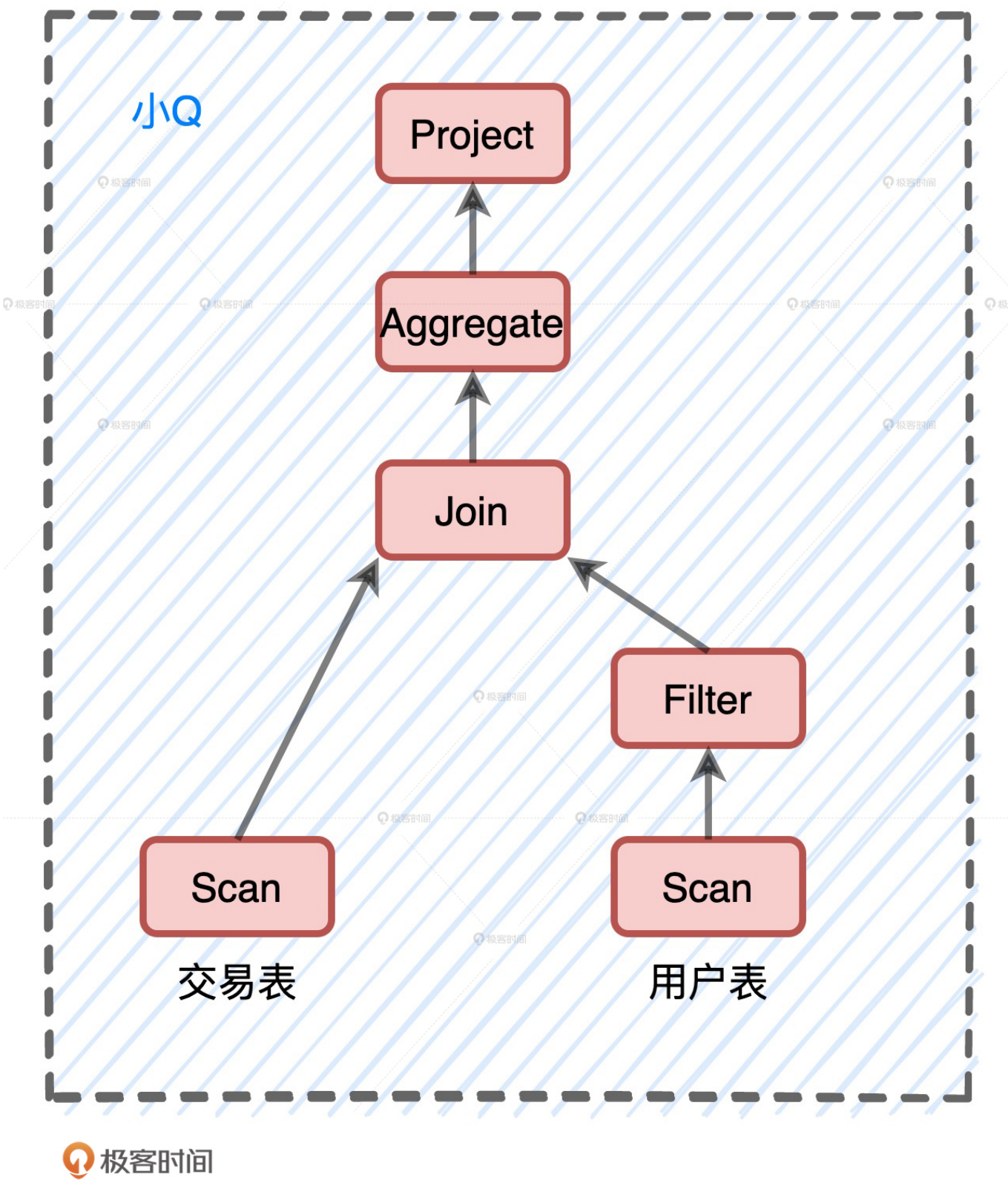
我们先来看一个例子，例子来自电子商务场景，业务需求很简单：给定交易事实表 transactions 和用户维度表 users，统计不同用户的交易额，数据源以 Parquet 的格式存储在分布式文件系统。因此，我们要先用 Parquet API 读取源文件。

复制代码

```
val userFile: String = _
val usersDf = spark.read.parquet(userFile)
usersDf.printSchema
/**
root
|-- userId: integer (nullable = true)
|-- name: string (nullable = true)
|-- age: integer (nullable = true)
|-- gender: string (nullable = true)
|-- email: string (nullable = true)
*/
val users = usersDf
.select("name", "age", "userId")
.filter($"age" < 30)
.filter($"gender".isin("M"))

val txFile: String = _
val txDf = spark.read.parquet(txFile)
txDf.printSchema
/**
root
|-- itemId: integer (nullable = true)
|-- userId: integer (nullable = true)
|-- price: float (nullable = true)
|-- quantity: integer (nullable = true)
*/

```

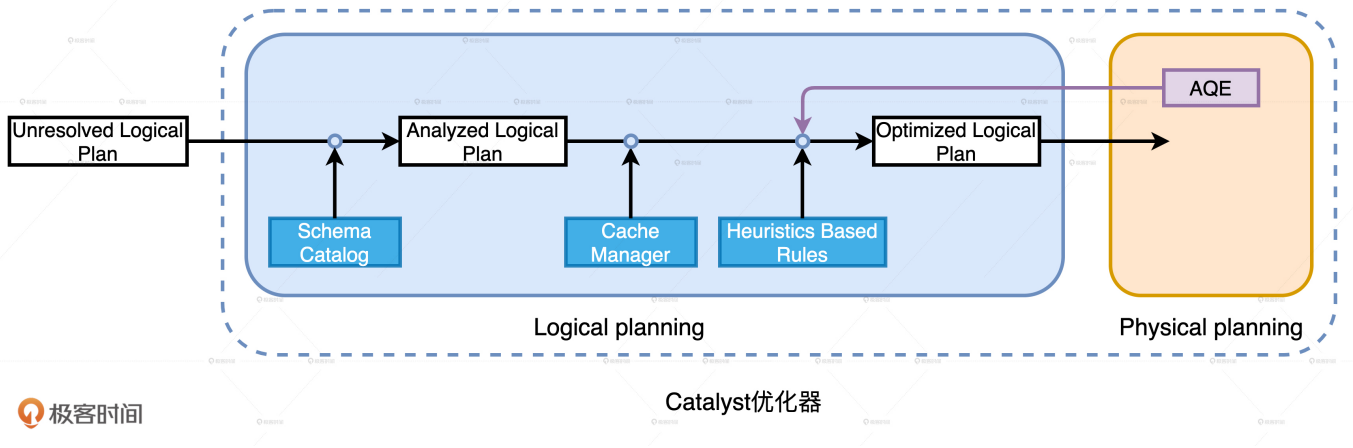
```
val result = txDF.select("price", "volume", "userId")
.join(users, Seq("userId"), "inner")
.groupBy(col("name"), col("age")).agg(sum(col("price") * col("volume")).alias(

result.write.parquet("_")
```

代码示例如上图所示，为了实现业务逻辑，我们对过滤之后的用户表与交易表做内关联，然后再按照用户分组去计算交易额。不难发现，这个计算逻辑实际上就是星型数仓中典型的关联查询。为了叙述方便，我们给这个关联查询起个名字：小 Q。小 Q 的计算需要两个输入源，一个是交易表，另一个是过滤之后的用户表。今天这一讲，我们就去追随小 Q，看看它在 Catalyst 的逻辑优化阶段都会发生哪些变化。



小Q的计算逻辑

Catalyst 逻辑优化阶段分为两个环节：逻辑计划解析和逻辑计划优化。在逻辑计划解析中，Catalyst 把“Unresolved Logical Plan”转换为“Analyzed Logical Plan”；在逻辑计划优化中，Catalyst 基于一些既定的启发式规则（Heuristics Based Rules），把“Analyzed Logical Plan”转换为“Optimized Logical Plan”。



Catalyst逻辑优化阶段

因为“Unresolved Logical Plan”是 Catalyst 优化的起点，所以在进入 Catalyst 优化器之前，小 Q 先是改头换面，从代码中的查询语句，摇身变成了“Unresolved Logical Plan”。

```
== Parsed Logical Plan ==
Join Inner, (userId#56 = userId#66)
:- Project [userId#56, name#57, age#58]
:  +- Filter gender#59 IN (M)
:     +- Filter (age#58 < 30)
:        +- Project [userId#56, name#57, age#58, gender#59]
:           +- Relation[userId#56,name#57,age#58,gender#59,email#60] parquet
+- Relation[userId#66,itemId#67,price#68,quantity#69] parquet
```

小Q启程：Unresolved Logical Plan

逻辑计划解析

小 Q 成功进入 Catalyst 优化器之后，就要开始执行逻辑计划解析，也就是要从“Unresolved Logical Plan”转换为“Analyzed Logical Plan”。那么，具体该怎么做呢？

从“小 Q 启程”那张图我们不难发现，“Unresolved Logical Plan”携带的信息相当有限，它只包含查询语句从 DSL 语法变换成 AST 语法树的信息。需要说明的是，不论是逻辑计划还是物理计划，执行的次序都是自下向上。因此，图中逻辑计划的计算顺序是从全表扫描到按性别过滤，每个步骤的含义都是准备“做什么”。

例如，在计划的最底层，Relation 节点“告诉”Catalyst：“你需要扫描一张表，这张表有 4 个字段，分别是 ABCD，文件格式是 Parquet”。但这些信息对于小 Q 的优化还远远

不够，我们还需要知道这张表的 Schema 是啥？字段的类型都是什么？字段名是否真实存在？数据表中的字段名与计划中的字段名是一致的吗？

因此，**在逻辑计划解析环节，Catalyst 就是要结合 DataFrame 的 Schema 信息，来确认计划中的表名、字段名、字段类型与实际数据是否一致。**完成确认之后，Catalyst 会生成“Analyzed Logical Plan”。这个时候，小 Q 就会从“Unresolved Logical Plan”转换成“Analyzed Logical Plan”。

从下图中我们能够看到，逻辑计划已经完成了一致性检查，并且可以识别两张表的字段类型，比如 userId 的类型是 int，price 字段的类型是 double 等等。

```
== Analyzed Logical Plan ==
userId: int, name: string, age: int, userId: int, itemId: int, price: double, quantity: int
Join Inner, (userId#56 = userId#66)
:- Project [userId#56, name#57, age#58]
:  +- Filter gender#59 IN (M)
:     +- Filter (age#58 < 30)
:        +- Project [userId#56, name#57, age#58, gender#59]
:           +- Relation[userId#56,name#57,age#58,gender#59,email#60] parquet
+- Relation[userId#66,itemId#67,price#68,quantity#69] parquet
```

小Q再变身：Analyzed Logical Plan

逻辑计划优化

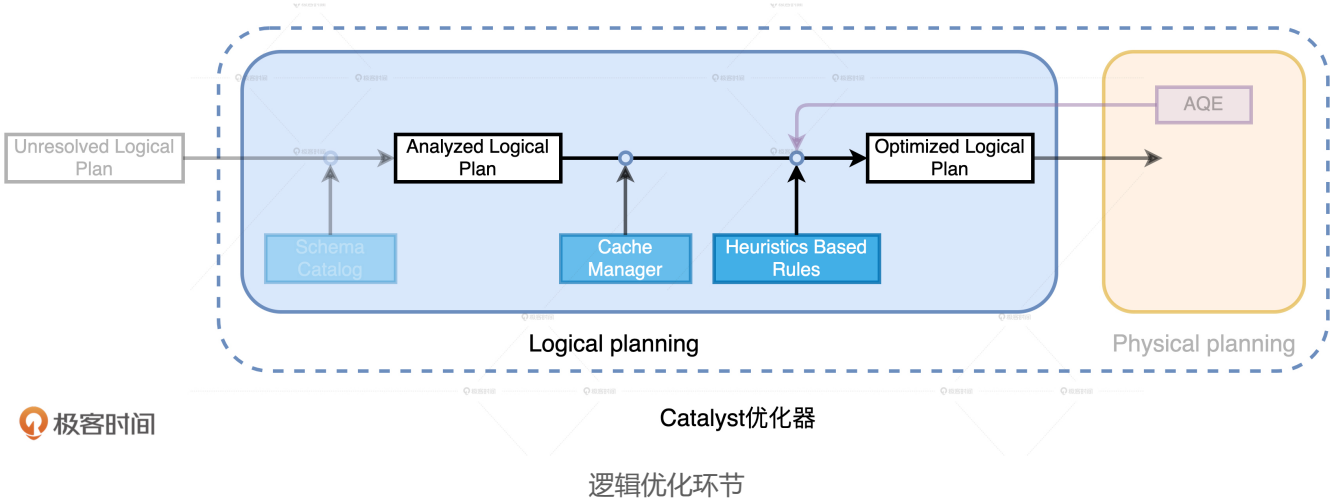
对于现在的小 Q 来说，如果我们不做任何优化，直接把它转换为物理计划也可以。但是，这种照搬开发者的计算步骤去制定物理计划的方式，它的执行效率往往不是最优的。

为什么这么说呢？在运行时，Spark 会先全量扫描 Parquet 格式的用户表，然后遴选出 userId、name、age、gender 四个字段，接着分别按照年龄和性别对数据进行过滤。

对于这样的执行计划来说，最开始的全量扫描显然是一种浪费。原因主要有两方面：一方面，查询实际上只涉及 4 个字段，并不需要 email 这一列数据；另一方面，字段 age 和 gender 上带有过滤条件，我们完全可以利用这些过滤条件减少需要扫描的数据量。

由此可见，**对于同样一种计算逻辑，实现方式可以有多种，按照不同的顺序对算子做排列组合，我们就可以演化出不同的实现方式。最好的方式是，我们遵循“能省则省、能拖则拖”的开发原则，去选择所有实现方式中最优的那个。**

同样，在面对这种“选择题”的时候，Catalyst 也有一套自己的“原则”和逻辑。因此，生成“Analyzed Logical Plan”之后，Catalyst 并不会止步于此，它会基于一套启发式的规则，把“Analyzed Logical Plan”转换为“Optimized Logical Plan”。



逻辑优化环节

那么问题来了，Catalyst 都有哪些既定的规则和逻辑呢？基于这些规则，Catalyst 又是怎么做转换的呢？别着急，我们一个一个来解答，咱们先来说说 Catalyst 的优化规则，然后再去探讨逻辑计划的转换过程。

Catalyst 的优化规则

和 Catalyst 相比，咱们总结出的开发原则简直就是小巫见大巫，为什么这么说呢？在新发布的 Spark 3.0 版本中，Catalyst 总共有 81 条优化规则（Rules），这 81 条规则会分成 27 组（Batches），其中有些规则会被收纳到多个分组里。因此，如果不考虑规则的重复性，27 组算下来总共会有 129 个优化规则。

对于如此多的优化规则，我们该怎么学呢？实际上，如果从优化效果的角度出发，这些规则可以归纳到以下 3 个范畴：

谓词下推（Predicate Pushdown）

列剪裁（Column Pruning）

常量替换 （Constant Folding）

首先，我们来说说谓词下推谓词下推主要是围绕着查询中的过滤条件做文章。其中，**“谓词”指代的是像用户表上“age < 30”这样的过滤条件，“下推”指代的是把这些谓词沿**

着执行计划向下，推到离数据源最近的地方，从而在源头就减少数据扫描量。换句话说，让这些谓词越接近数据源越好。

不过，在下推之前，Catalyst 还会先对谓词本身做一些优化，比如像 OptimizeIn 规则，它会把“gender in ‘M’ ”优化成“gender = ‘M’ ”，也就是把谓词 in 替换成等值谓词。再比如，CombineFilters 规则，它会把“age < 30”和“gender = ‘M’ ”这两个谓词，捏合成一个谓词：“age != null AND gender != null AND age <30 AND gender = ‘M’ ”。

完成谓词本身的优化之后，Catalyst 再用 PushDownPredicte 优化规则，把谓词推到逻辑计划树最下面的数据源上。对于 Parquet、ORC 这类存储格式，结合文件注脚（Footer）中的统计信息，下推的谓词能够大幅减少数据扫描量，降低磁盘 I/O 开销。

再来说说列剪裁。**列剪裁就是扫描数据源的时候，只读取那些与查询相关的字段。**以小 Q 为例，用户表的 Schema 是（userId、name、age、gender、email），但是查询中压根就没有出现过 email 的引用，因此，Catalyst 会使用 ColumnPruning 规则，把 email 这一列“剪掉”。经过这一步优化，Spark 在读取 Parquet 文件的时候就会跳过 email 这一列，从而节省 I/O 开销。

不难发现，谓词下推与列剪裁的优化动机，其实和“能省则省”的原则一样。核心思想都是用尽一切办法，减少需要扫描和处理的数据量，降低后续计算的负载。

最后一类优化是常量替换，它的逻辑比较简单。假设我们在年龄上加的过滤条件是“age < 12 + 18”，Catalyst 会使用 ConstantFolding 规则，自动帮我们把条件变成“age < 30”。再比如，我们在 select 语句中，掺杂了一些常量表达式，Catalyst 也会自动地用表达式的结果进行替换。

到此为止，咱们从功用和效果的角度，探讨了 Catalyst 逻辑优化规则的 3 大范畴。你可能说：“拢共就做了这么 3 件事，至于兴师动众地制定 81 条规则吗？”我们划分这 3 大范畴，主要是为了叙述和理解上的方便。实际上，对于开发者写出的五花八门、千奇百怪的查询语句，正是因为 Catalyst 不断丰富的优化规则，才让这些查询都能够享有不错的执行性能。如果没有这些优化规则的帮忙，小 Q 的执行性能一定会惨不忍睹。

最终，被 Catalyst 优化过后的小 Q，就从“Analyzed Logical Plan”转换为“Optimized Logical Plan”，如下图所示。我们可以看到，谓词下推和列剪裁都体现到了 Optimized Logical Plan 中。

```
== Optimized Logical Plan ==
Join Inner, (userId#56 = userId#66)
:- Project [userId#56, name#57, age#58]
:  +- Filter ((((isnotnull(age#58) AND isnotnull(gender#59))
:             AND (age#58 < 30)) AND (gender#59 = M)) AND isnotnull(userId#56))
:     +- Relation[userId#56,name#57,age#58,gender#59,email#60] parquet
+- Filter isnotnull(userId#66)
   +- Relation[userId#66,itemId#67,price#68,quantity#69] parquet
```

小Q再变身：Optimized Logical Plan

Catalys 的优化过程

接下来，我继续来回答刚刚提出的第二个问题：基于这么多优化规则，Catalyst 具体是怎么把“Analyzed Logical Plan”转换成“Optimized Logical Plan”的呢？其实，不管是逻辑计划（Logical Plan）还是物理计划（Physical Plan），它们都继承自 QueryPlan。

QueryPlan 的父类是 TreeNode，TreeNode 就是语法树中对于节点的抽象。TreeNode 有一个名叫 children 的字段，类型是 Seq[TreeNode]，利用 TreeNode 类型，Catalyst 可以很容易地构建一个树结构。

除了 children 字段，TreeNode 还定义了很多高阶函数，其中最值得关注的是一个叫做 transformDown 的方法。transformDown 的形参，正是 Catalyst 定义的各种优化规则，方法的返回类型还是 TreeNode。另外，transformDown 是个递归函数，参数的优化规则会先作用（Apply）于当前节点，然后依次作用到 children 中的子节点，直到整棵树的叶子节点。

总的来说，从“Analyzed Logical Plan”到“Optimized Logical Plan”的转换，就是从一个 TreeNode 生成另一个 TreeNode 的过程。Analyzed Logical Plan 的根节点，通过调用 transformDown 方法，不停地把各种优化规则作用到整棵树，直到把所有 27 组规则尝试完毕，且树结构不再发生变化为止。这个时候，生成的 TreeNode 就是 Optimized Logical Plan。

为了把复杂问题简单化，我们使用 Expression，也就是表达式来解释一下这个过程。因为 Expression 本身也继承自 TreeNode，所以明白了这个例子，TreeNode 之间的转换我们也就清楚了。

复制代码

```
//Expression的转换
import org.apache.spark.sql.catalyst.expressions._
val myExpr: Expression = Multiply(Subtract(Literal(6), Literal(4)), Subtract(L
val transformed: Expression = myExpr transformDown {
  case BinaryOperator(l, r) => Add(l, r)
  case IntegerLiteral(i) if i > 5 => Literal(1)
  case IntegerLiteral(i) if i < 5 => Literal(0)
}
```

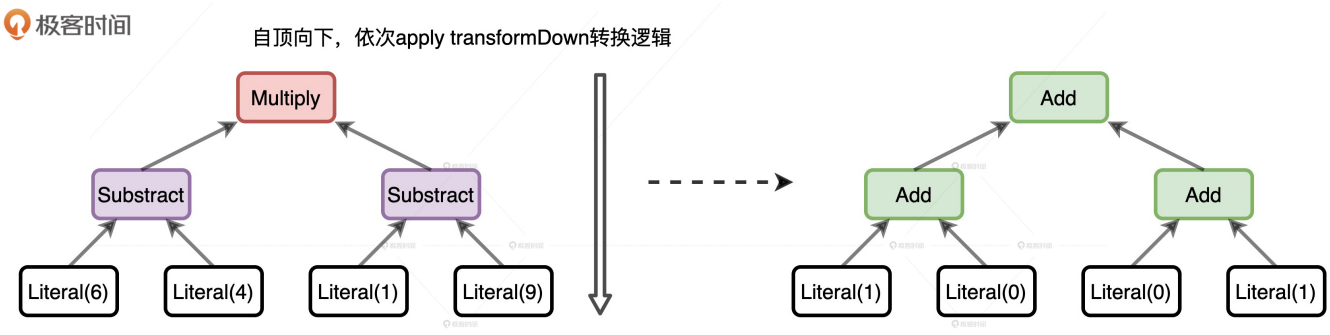
首先，我们定义了一个表达式：（（6 - 4）*（1 - 9）），然后我们调用这个表达式的 transformDown 高阶函数。在高阶函数中，我们提供了一个用 case 定义的匿名函数。显然，这是一个偏函数（Partial Functions），你可以把这个匿名函数理解成“自定义的优化规则”。在这个优化规则中，我们仅考虑 3 种情况：

对于所有的二元操作符，我们都把它转化成加法操作

对于所有大于 5 的数字，我们都把它变成 1

对于所有小于 5 的数字，我们都把它变成 0

虽然我们的优化规则没有任何实质性的意义，仅仅是一种转换规则而已，但是这并不妨碍你去理解 Catalyst 中 TreeNode 之间的转换。当我们把这个规则应用到表达式（（6 - 4）*（1 - 9））之后，得到的结果是另外一个表达式（（1 + 0）+（0 + 1）），下面的示意图直观地展示了这个过程。



自顶向下对执行计划进行转换

从“Analyzed Logical Plan”到“Optimized Logical Plan”的转换，与示例中表达式的转换过程如出一辙。最主要的区别在于，Catalyst 的优化规则要复杂、精密得多。

Cache Manager 优化

从“Analyzed Logical Plan”到“Optimized Logical Plan”的转换，Catalyst 除了使用启发式的规则以外，还会利用 Cache Manager 做进一步的优化。

这里的 Cache 指的就是我们常说的分布式数据缓存。想要对数据进行缓存，你可以调用 DataFrame 的.cache 或.persist，或是在 SQL 语句中使用“cache table”关键字。

Cache Manager 其实很简单，它的主要职责是维护与缓存有关的信息。具体来说，Cache Manager 维护了一个 Mapping 映射字典，字典的 Key 是逻辑计划，Value 是对应的 Cache 元信息。

当 Catalyst 尝试对逻辑计划做优化时，会先尝试对 Cache Manager 查找，看看当前的逻辑计划或是逻辑计划分支，是否已经被记录在 Cache Manager 的字典里。如果在字典中可以查到当前计划或是分支，Catalyst 就用 InMemoryRelation 节点来替换整个计划或是计划的一部分，从而充分利用已有的缓存数据做优化。

小结

今天这一讲，我们主要探讨了 Catalyst 优化器的逻辑优化阶段。这个阶段包含两个环节：逻辑计划解析和逻辑计划优化。

在逻辑计划解析环节，Catalyst 结合 Schema 信息，对于仅仅记录语句字符串的 Unresolved Logical Plan，验证表名、字段名与实际数据的一致性。解析后的执行计划称为 Analyzed Logical Plan。

在逻辑计划优化环节，Catalyst 会同时利用 3 方面的力量优化 Analyzed Logical Plan，分别是 AQE、Cache Manager 和启发式的规则。它们当中，Catalyst 最倚重的是启发式的规则。

尽管启发式的规则多达 81 项，但我们把它们归纳为 3 大范畴：谓词下推、列剪裁和常量替换。我们要重点掌握谓词下推和列剪裁，它们的优化动机和“能省则省”的开发原则一

样，核心思想都是用尽一切办法，减少需要扫描和处理的数据量，降低后续计算的负载。

针对所有的优化规则，Catalyst 优化器会通过调用 TreeNode 中的 transformDown 高阶函数，分别把它们作用到逻辑计划的每一个节点上，直到逻辑计划的结构不再改变为止，这个时候生成的逻辑计划就是 Optimized Logical Plan。

最后，Cache Manager 的作用是提供逻辑计划与数据缓存的映射关系，当现有逻辑计划或是分支出现在 Cache Manager 维护的映射字典的时候，Catalyst 可以充分利用已有的缓存数据来优化。

每日一练

1. 既然 Catalyst 在逻辑优化阶段有 81 条优化规则，我们还需要遵循“能省则省、能拖则拖”的开发原则吗？
2. 你能说说 Spark 为什么用偏函数，而不是普通函数来定义 Catalyst 的优化规则吗？

期待在留言区看到你的思考和答案，我们下一讲见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 20 | RDD和DataFrame：既生瑜、何生亮

下一篇 22 | Catalyst物理计划：你的SQL语句是怎么被优化的（下）？

精选留言 (6)

写留言

kingcall
2021-04-30

问题1：开发阶段要有极客精神，尽量将优化实现在自己的代码里，而不是依赖框架，因为框架是一个普世的优化，还有就是如果我们根据业务特点进行了优化再加上框架本身带来的优化能给我们的程序带来一个更好的性能提升,也就是说上层自我优化和底层框架优化。
问题2：与偏函数对应的一个定义就是我们数学意义上的函数，一个输入自变量x对应一个输出因变量y，也就是y 可以表示成x 的一个特定的运算，并且这个运算关系是确定的，...
展开

作者回复: Perfect x 2！标准答案了~ 💯

回答的非常好了，无可挑剔~

1 9

对方正在输入。。。
2021-05-01

老师的课，我是越看越爽，看完就有一种“老子天下无敌了”的感觉，哈哈哈哈哈

展开

作者回复: 哈哈哈，喜欢就好~ 兄弟五一节快乐~

1

sky_sql
2021-05-01

老师好，RDD api有点类似MR编程，Spark SQL有点类似hive，过程都包括使用 Antlr 实现 SQL 的词法和语法解析，后面也有Schema 信息验证，优化环节谓词下推、列剪裁等?

展开

作者回复: 对，构建语法树、Schema验证、谓词下推、列剪裁，这些其实是绝大多数的数仓都有的优化策略。

实际上，相比传统DBMS，Spark SQL的优化流程算是简化版的了，比如传统DBMS还会有Query Re-writer、CBO（基于成本的优化）等等。

1

GEEKBANG_2743065
2021-04-30

1. 既然 Catalyst 在逻辑优化阶段有 81 条优化规则，我们还需要遵循“能省则省、能拖则拖”的开发原则吗？你能说说 Spark 为什么用偏函数，而不是普通函数来定义 Catalyst 的优化规则吗?
答：要。战略上用开发原则，战术上依赖Catalyst。
...

展开

作者回复: 第二题分析的很到位，就是你说的这么回事。偏函数的特性，刚好符合Spark SQL优化规则的特点，也就是：
1. 不完备
2. 保持扩展的灵活性

问题三是个超级好的问题，其实我一直想找机会说说。在传统的DBMS里，CBO是主流，RBO是辅助性的。原因其实很简单，RBO毕竟是出于启发式的，都是一些“经验主义”，CBO才是真正“尊重”事实、尊重运行时的优化策略。在传统DBMS里面，CBO是非常成熟的技术了，已经沿用很多很多年了。

然而，Spark SQL的CBO相对就比较鸡肋了，原因其实我在第24讲会详细展开，提前剧透一下，就是Spark SQL的CBO有三个非常大的痛点：窄、慢、静

窄：窄指的是适用面太窄，CBO仅支持注册到Hive Metastore的数据表，但在大量的应用场景中，数据源往往是存储在分布式文件系统的各类文件，如Parquet、ORC、CSV等等。

慢：慢指的是统计信息的搜集效率比较低。对于注册到Hive Metastore的数据表，开发者需要调用ANALYZE TABLE COMPUTE STATISTICS语句收集统计信息，而各类信息的收集会消耗大量时间。

静：静指的是静态优化，这一点与RBO一样。CBO结合各类统计信息制定执行计划，一旦执行计划交付运行，CBO的使命就算完成了。这一点和传统DBMS很不同，传统DBMS的CBO是动态的，可以在运行时做适当调整。

不仅如此，目前Spark SQL的CBO，还仅仅支持Join策略，换句话说，与Join无关的查询，CBO使不上劲。

因此，综上，CBO虽然是个非常吸引人的东西，但是Spark SQL的CBO很鸡肋，当然这有很多原因，比如CBO本身的限制，社区对于Spark SQL优化方向的考虑，等等。其实AQE的推出，就表示Spark社区已经做出了选择。就我个人观察（仅代表个人观点哈），CBO的地位和角色可能会越来越尴尬，用武之地可能会越来越小，聊胜于无。当然，再次重申，这仅仅是我个人的观点哈~

另外，这两个问题kingcall同学回答的比较好，可以参考他的答案哈~

1 1

aof

2021-05-05

偏函数只针对部分输入来输出结果，而每个函数对应的优化规则也是有限的，再搭配模式匹配，很完美的应用场景哈哈

展开

作者回复: 对，没错，完美契合！老弟这进度赶得飞起啊~ 都追到21讲了~ 赞

zxk

2021-05-01

问题1：Spark 里的优化只是一种普世的优化，力求在尽可能覆盖绝大部分通用场景，减少对开发者的要求。但实际开发中场景千差万别，Spark 不一定能覆盖到所有场景，因此仍需要我们尽可能遵循这些开发原则。
问题2：对偏函数不是很熟悉，猜测是尽可能的拆解函数粒度，给 Spark 留出更多的优化空间。比如之前老师就有个 label encoding 的例子，将函数拆分为偏函数后，Spark 自...

展开

作者回复: 第一个问题答得很好~
第二个问题可以参考kingcall同学的答案哈~